Exploring Advanced Techniques for Training an Artificial Neural Network on a Complex, Multimodal Game Scenario

Ethan Bogdan
Swarthmore College

Abstract

I conduct a comparative study of different techniques (standard backprop, complementary reinforcement backprop, NEAT) for training a recurrent neural net on a difficult problem. I contribute an additional layer of strategy by offering the net access to expert algorithms, which can provide domain knowledge about the task at hand. My hypothesis was that learning how to make use of this expert knowledge would prove an easier task than learning the problem from scratch. Results support this hypothesis, but indicate that experts also introduce further complication.

Artificial neural network (ANNs), inspired by the biological circuitry of the human brain, are one of the oldest and most versatile algorithms for machine learning. Academics and industry professionals have successfully applied them to a wide variety of learning tasks, spanning the gamut from facial recognition to e-mail spam filtering and more.

By and large, the basic structure of neural nets has remained the same over many years.[1] However, effective training of a neural net can require a fair amount of wrangling, and different approaches have proven most successful in different contexts. I am interested in pushing the limits of neural nets by placing them in a context where they have traditionally done poorly, and then exploring new options for training.

The context I chose for this study is a puzzle game called Mastermind. It presents two primary challenges as a learning task:

1. Good moves require a memory of all previous moves that have been made
2. Optimal strategy changes as the game progresses

In theory, neural nets with "recurrence" – that is, connections which relay information backwards – are capable of factoring previous decisions into future ones. However, the depth and persistence of this kind of memory is a subject of ongoing research, and neural nets tend to perform best when short-term memory is unimportant.

[1]Recent advances in "deep learning" are the most notable exception.

Neural nets can also suffer from the issue of "catastrophic forgetting" – that is, even their long-term memory can fail, provided significant changes in environmental conditions (like the shifting game strategies of Mastermind). Some research has tried to address this problem by performing clustering analysis, and forking off different local experts to learn each cluster.[2]

The defining concept for this study was to apply a local-expert-based model to a recurrent neural network. However, rather than evolving experts from scratch, I wanted to determine whether the network could learn to consult pre-established experts in its domain. These experts were hard-coded with a variety of standard strategies for playing Mastermind. None of them could play an entire game in an optimal fashion, but each possessed some knowledge that might be useful during particular game phases. It is worth noting that Mastermind was an especially convenient candidate for this kind of supervised learning, because of how easy it is to verify, for any given move, which expert would have offered the best advice.

This study has potential implications both for training ANNs on multimodal learning tasks in general, and for integrating pre-existing domain knowledge into the learning process. Ideally, it would be possible to use these techniques to take experts trained in CBIM (or some equivalent algorithm) and apply them to new learning contexts. My work also offers promise for making optimal use of redundant control systems – for example, programming an aircraft to automatically know when to trust its pilot and when to trust its co-pilot instead. Multiple experts are only better than one provided that they are each allowed to play to their respective strengths. A well-trained ANN might be able to classify exactly what those strengths are and when to rely on them.

Background

A previous study by Lisa Meeden considers the problem of training ANNs (or "connectionist systems") on tasks that require some temporal planning. (Meeden 1996) She draws a helpful distinction between *local* techniques (like reinforcement learning) and *global* techniques (like genetic al-

[2]c.f. Intelligent Adaptive Curiosity (which uses a form of k-nearest-neighbors), Category-Based Intrinsic Motivation (which uses a Growing Neural Gas), and Horde

gorithms) for learning. This framework guides my current angles of approach, and, in particular, I have adopted a very similar style of reinforcement learning, via complementary reinforcement back-propagation (CRBP). However, Meeden's work emphasizes an ANN's essential nature as a bottom-up strategy, while I would like to explore a closer marriage between hands-off, bottom-up evolution and the utilization of top-down domain knowledge. Perhaps the experts in my study fulfill the role of Waltz's "control structures [that] allow planning," a higher-level component of artificial intelligence, which Meeden's study does not explore.

In terms of learning to apply expert knowledge to new domains, some promising groundwork has already been conducted using recurrent neural nets, indicating that they can make generalizations about patterns of letters based only on prior exposure to analogous patterns of audio data. (Dienes, Altmann, and Gao 1999) Unfortunately, the authors showed that this transfer of knowledge depended on adding an extra hidden layer to their Elman network, which will be beyond the scope of my current study.

Experimental setup

I conducted all of the coding for this lab in Python, which allowed for fast development, while simultaneously offering a sufficiently speedy runtime environment to train my networks on tens of thousands of data points. I wrote a custom Mastermind class from scratch, but I relied heavily on Conx (packaged with PyroRobot) for back-propagation and a standard Python implementation of NEAT for genetic evolution. In this section, I will discuss each of these components in further depth.

Mastermind

In its earliest incarnation, Mastermind was a physical boardgame invented in 1970. At the start of the game, one player would come up with a secret code, represented by a series of colored pegs placed on a board and hidden behind a divider. Subsequent play would involve the other player trying to match this code by placing their own pegs on the board. After each guess, the first player would respond with a number of black and white pegs indicating, respectively, how many of the guess pegs were present somewhere in their secret code, and how many of them were matched in the correct position.

The difficulty of this game depends on how long the secret code is (the puzzle size) and how many different kinds of symbols it could contain (the alphabet size). For the purposes of this study, I set both of these parameters to four and used a numeric alphabet. This meant that there were $4^4 = 256$ valid guesses, represented by the set (0000, 0001, ... 3332, 3333). For a human with complete memory of game history, this setup presents a very easy challenge, typically solvable within at most 10 guesses. For a neural net, though, collating data collected over the course of 10 activations is a non-trivial task.

Strategy

Mastermind lends itself to a number of different styles of play, which I attempted to distill into three expert heuristics

Figure 1: Screenshot of a basic Mastermind GUI; note that there are 6 colors in the alphabet and that the puzzle size is 4

(along with one "bad" player). Each one takes a previous guess as input, then transforms it into a new guess at what the secret code might be. The heuristics are as follows:

Incremental change expert

Implementation: Randomize one symbol in code, keeping the rest the same
Notes: Expected to be a decent strategy for the mid to later part of the game, when we are closing in on the right answer

Swapping expert

Implementation: Randomly choose two symbols in code and swap them
Notes: Expected to be useful when the characters in the previous guess are mostly correct, but appear in the wrong order; expected to perform poorly when the previous guess contains many duplicate characters, since random swapping may have no effect

Non-overlapping expert

Implementation: Generate new code randomly from alphabet of symbols not used in the previous guess (e.g. '2232' $\rightarrow$ '1001'); if all symbols were used, then just generate a code with no positional matches (e.g. '0123' $\rightarrow$ '2300')
Notes: Expected to be most useful early in the game, when making diverse guesses can help narrow the search space faster

Random player

Implementation: Generate entirely random string from alphabet, disregarding previous guess
Notes: This "expert" serves as both a baseline and a red herring; my expectation was that the neural net would learn never to consult it.

Optimal play

In order to perform local reinforcement learning, it was necessary to establish a metric for determining which of these experts was the "right" expert to trust on each turn of the game. If there were an obvious answer to this question, Mastermind would present little value as a learning task. Fortunately, while there is rarely an obvious answer, there does exist a straightforward and robust method for evaluating the merit of any given guess, after that guess has been made.

This method involves maintaining a list of all potential candidates for the secret code. Every time the player makes a guess – and gets feedback about the number of correct characters, number of characters in correct positions, etc. – the algorithm runs through the list of remaining secret code candidates. Codes which would have yielded the same feedback if they were the secret code are kept in the list, while any codes that would have yielded different feedback are removed. In effect, this method implements a simple form of constraint propagation, such that, after a handful guesses, only one or two possibilities will remain. By calculating the number of potential codes that each guess rules out, we can assign a numerical value to the utility of that guess.

In my code, I determined optimal play based on two factors. The first was this constraint-propagation technique, while the second simply looked at how similar a guess was to the correct answer. For example, if swapping the characters in a code wouldn't have ruled out many potential candidates, but would nonetheless have resulted in two more characters being in the correct positions, I wanted to reinforce that behavior. The details of how this reinforcement worked are discussed under "Local training" below.

Network topology

Neural nets consist of a collection of nodes (representing neurons) and edges (representing synapses), interconnected in any number of different configurations. Typically, there are at least one layer of nodes serving as inputs, one layer serving as outputs, and one or more "hidden" layers in between.

For the purposes of local training, I used a fixed topology with 18 input nodes, 12 hidden nodes, and 3 output nodes. These three layers were fully connected (input to hidden, hidden to output), with the middle (hidden) layer having a recurrent relationship with itself – the standard structure for an Elman network. Among the input nodes, there were 16 nodes that represented the previous guess the player had made and two representing the feedback that that guess had generated (i.e. percentage of characters correct and percentage of characters in the correct place). Each of the output nodes indicated a decision to trust a given expert. After activation of the network, a winner-takes-all process would determine which of the outputs was dominant, and then the corresponding expert would provide the next guess for the player to use. (See Fig. 2, below.)

My primary global training involved the same set of inputs, but with a fourth output node also present, which corresponded to the random player. This allowed me to test the hypothesis that networks which ignored the fourth node would have more evolutionary fitness than networks which sometimes made random guesses. The rest of the topology (hidden nodes, edges etc.) was variable, as discussed under "Global training" below.

Finally, for the sake of comparison, I also studied one additional global training scenario, in which I discarded the experts altogether and attempted to learn Mastermind from scratch – a seemingly monumental task. Once again, the input nodes were the same as those used for local training, but there were now 16 output nodes in four groups of four, which encoded an output guess in the same manner as the input nodes encoded the previous, input guess. (For a representation of this general encoding, see bottom of Fig 2.; outputs, per usual, were resolved via winner takes all.)

Local training

The general premise of local training is for our neural network to receive feedback at each step of the way as it plays Mastermind. After each new set of inputs propagates forwards through the net, there will be a corresponding backpropagation of targets and updating of edge weights. By embodying a very close linkage of cause and effect, local training can theoretically support more nuanced and closely targeted learning than global training. However, it also risks being short-sighted, or producing solutions that don't generalize very well.

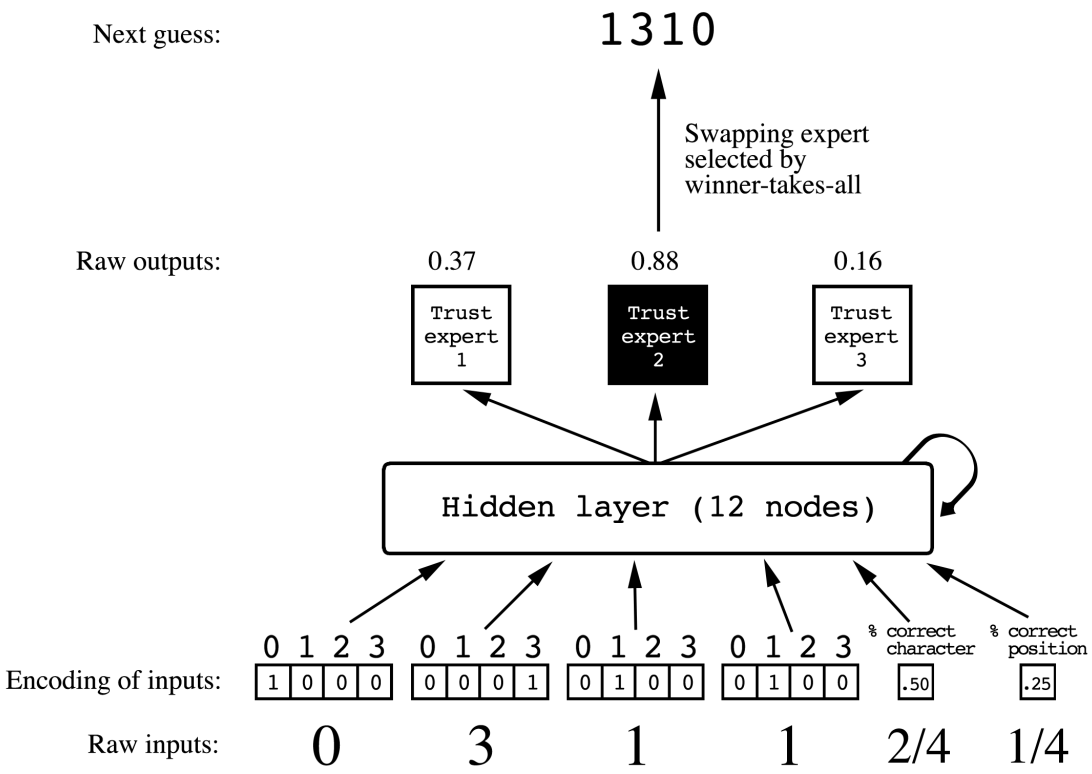
This study involved three variant methods of determining targets for local training, all of which utilized the scoring data discussed above (see "Optimal play"):

- *Winner Takes All*
  This approach reinforces only the expert that would have yielded the maximal score. For example, for scoring vector $S = \{19, 112, 50\}$, the corresponding target vector would be $T = \{0, 1, 0\}$. Ties get settled randomly.

- *Gradient Targets*
  Rather than reducing the scoring vector to a completely binary representation, this approach merely scales it to fit within the range [0,1]. As a simple example, scoring vector $S = \{200, 150, 100\}$ becomes target vector $T = \{1, 0.5, 0\}$. In the rare case that all scores are equal, this method returns $T = \{0.5, 0.5, 0.5\}$.

- *Complementary Reinforcement*
  Unlike the previous two methods, complementary reinforcement takes account of which expert the network has already decided to trust. If this expert yields a score surpassing a certain threshold, then complementary reinforcement rewards the network for choosing that expert, in proportion to the strength of that score. [3] Otherwise, if the score is below said threshold, complementary reinforcement punishes the network by offering small rewards to the other two experts, and no reward to the

[3] Because scores get lower in the later stages of a game, I normalized this value by dividing it by the maximum possible score for the given turn.



Secret code: `2330`
Previous guess: `0311`
Character score: `2/4`
Position score: `1/4`

Figure 2: Diagram of network topology for local reinforcement, with example data

chosen one. Only the score of the chosen expert has any bearing on this process.

Regardless of approach, local learning entailed plugging these targets into the standard back-propagation algorithm, with a learning rate of 0.3 and a momentum of 0.05. Games were allowed to run for at most 15 guesses, or until the player had won, whichever came sooner. The total training time for each local learning method was 10,000 guesses.

Global training

Many of the questions and subtleties associated with local training disappear when we adopt a global approach. The only relevant metric now is the actual, demonstrated success of an individual at playing Mastermind over the course of its "lifetime." How many games did it win? How long did it take for it to win them? This holistic view has the advantage of being firmly grounded in our end goals, without us, as developers, ever needing to know how best to achieve them. However, that lack of immediate guidance is exactly why global training can fail on complex tasks like playing Mastermind: it is impossible to reward specific good behaviors and punish bad ones. Furthermore, the search space is very large, and we have no guarantee of traversing it quickly.

Nevertheless, the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, developed by Kenneth O. Stanley at UCF, tackles these problems by starting with a highly simplified search space, then gradually working its way up to more complex solutions. NEAT is a genetic algorithm, which means that it breeds and evolves entire populations of Mastermind players, each represented by a separate neural net. These nets start out with direct linkages between input and output nodes, and then probabilistically gain intermediate nodes and connections as they evolve. The key step to configuring NEAT is to provide a relevant *fitness function*, which essentially determines how likely an individual is to pass its genes along to the next generation. For this study, I wanted to select for individuals that consistently beat Mastermind in the smallest number of guesses, and so my function was:

$$fitness = \frac{maxGuesses - actualGuesses}{maxGuesses}$$

In other words, a player that always guesses the secret code on its first try will have a fitness of around 1, while a player that always exhausts the allowable number of guesses will have a fitness of 0. In order to allow individuals a better opportunity to differentiate themselves, I set a high upper limit of 1000 guesses per game (after which point the Mastermind class would reset itself). Then, in order to produce more robust results and minimize the role of dumb luck, I required each individual to play 10 games before receiving a fitness score. This meant that $maxGuesses = 1000\frac{\text{guesses}}{\text{game}} * 10$ games $= 10000$ guesses.

The only other setup required for running NEAT was to provide a config file with some basic evolutionary parameters. I used a population size of 40, with a species size of 10. Most of my evolutions ran for 200 generations (or "epochs"). Importantly, I set `feedforward = 0` in order to allow NEAT to evolve recurrent connections.

Player type	Mean guesses	Std. dev.
Optimal	96.857	96.471
Random	266.207	244.980
Random experts	316.510	288.606
Random "good" experts	402.917	332.565

Table 1: Benchmark data

Results and discussion

Benchmarks

In order to have a context in which to understand the effectiveness of my training, I ran a number of benchmark simulations, representing players at different extremes. The results are summarized in the following table: These data all refer to the number of guesses it takes to win a game of Mastermind, sampled over 1000 games. The "optimal" player selects experts based on constraint propagation; "random" ignores the experts and just makes random guesses; "random experts" trusts a different expert (including the random player) on each move; and "random 'good' experts" works similarly, except that it never chooses the random player.

Even among these benchmarks, there are several interesting findings. Most notably, listening to the expert algorithms at random actually results in significantly worse performance than ignoring the experts altogether! (Keep in mind that higher mean values indicate more guesses to win and thus lower fitness.) The random player traverses the search space uniformly, eventually hitting on the right code. The experts, however, have the potential to get stuck in various pockets of the space, or else to jump back and forth among a small set of bad guesses. Nonetheless, we see that it is definitely possible to obtain good results by making optimal use of the experts.

Local learning progress

By and large, local training did not prove an effective way to learn when to trust each of the expert algorithms. While complementary reinforcement back-propagation (CRBP) yielded marginally better results than the other approaches, all did worse than random:

Approach	Mean guesses	Std. dev.
CRBP	374.661	333.066
Gradient targets	390.061	344.606
Winner takes all	400.365	334.240

Table 2: Results for local training

Once again, these data were each collected by playing 1000 sample games after training had completed. Given their similarity in value to the "random 'good' experts" benchmark, it seems reasonable to assume that almost no

Approach	Trust Exp. 1	Trust Exp. 2	Trust Exp. 3
CRBP	33.4%	33.3%	33.3%
Gradient	33.4%	33.2%	33.4%
WTA	33.4%	33.3%	33.4%

Table 3: Expert trust rates for local training

meaningful learning has occurred.[4] Further support for this hypothesis resides in the following breakdown of time spent trusting each expert:

Although it is possible that all of the experts proved equally valuable, the utter uniformity of these percentages seems more strongly to suggest that the neural net has not learned to differentiate among the experts. There are many potential explanations for this failure of local training: perhaps the fixed topology was too simple; perhaps more hidden layers were necessary to preserve game context; perhaps the granularity of reinforcement was too fine and the patterns too abstract. The only obvious conclusion is that global training was the better approach.

Global learning progress

While most members of the first several generations repeatedly failed to guess the secret code, earning themselves fitness scores of 0, there were just enough successful individuals for fruitful evolution to get underway. In fact, learning progress was actually quite steep for the first 50 epochs or so, after which point the curve tended to level out, with an average fitness near 0.75 and a best fitness hovering around 0.9; (see Fig. 3). Below are results for a couple of the all-time best chromosomes:[5]

Epoch	Mean guesses	Std. dev.
187	237.518	235.498
198	246.977	237.212

Table 4: Best NEAT results

Although the standard deviation is high, these means fall significantly below the "random" benchmark – and even farther below the "random experts" benchmark. Thus global training has successfully gleaned some knowledge about each expert's strengths and weaknesses, and, more importantly, it has managed to use their collective domain knowledge to strategic advantage. (See Fig. 4 for a visualization of the network topology underlying this success.) A breakdown of how often each of these players trusts each expert offers further insight:

Encouragingly, the players have learned that some experts (notably Expert 1, the incremental change expert) are more helpful than others, and yet the evolved strategies do not neglect any of the experts altogether. This balance suggests a

[4]N.B.: Local training only had access to three output nodes, and thus could only choose from among the "good" experts; see "Network topology" above

[5]Each of these chromosomes comes from a different 200-epoch run of NEAT; they are not part of the same progression

Epoch	Expert 1	Expert 2	Expert 3	Random
187	40.6%	8.0%	5.2%	46.2%
198	31.3%	15.5%	2.4%	50.8%

Table 5: Expert trust rates for best NEAT chromosomes

nuanced representation of the problem. It is surprising to me that random play remains such a substantial element of strategy – around half of these moves are random – but I suspect that this is an important means of getting "unstuck" when experts offer circular advice. (Both Expert 2, the swapping expert, and Expert 3, the non-overlapping expert, can be guilty of cycling back and forth between recommended guesses.)

As a final piece of evidence that the task of learning Mastermind has benefited from expert knowledge, I present the results of training NEAT to play Mastermind from scratch:

Epoch	Mean guesses	Std. dev.
170	439.739	392.426
214	480.708	394.226

Table 6: Best results learning Mastermind from scratch

Over the course of a comparable number of epochs, NEAT hasn't learned to beat even the worst of the random benchmarks. This result may be expected, given the complex and multimodal nature of Mastermind, but it is especially significant in contrast to NEAT's relative success when consulting experts.

Further work

Although learning Mastermind from scratch was decidedly unsuccessful, it might be interesting to research a hybrid net that could choose between trusting an expert and doing its own learning, from scratch, on each turn. I suspect that having the freedom to output independent guesses might be especially helpful in the end game, once the net has already acquired enough knowledge about the secret code to outstrip the utility of its hard-coded expert consultants.

I would also like to find ways to make local training a more viable option. The Conx library does not seem to support recurrent networks with more than one hidden layer, but a more sophisticated topology might increase the capacity for remembering game history and detecting abstract patterns.

Finally, having established that it is indeed possible for a neural network to make effective use of domain knowledge gained elsewhere, I would like to test this hypothesis in more real-world scenarios. In particular, I'd like to see how well neural nets can learn to filter out advice from nefarious experts (for example, Internet "trolls" seeking to sabotage a game).

Conclusion

Introducing expert algorithms into a learning task can add new layers of complexity, even as it makes the problem at hand more accessible. Players naïvely trusting experts, with

insufficient knowledge of their relative merits, did worse than players who simply ignored the experts and took stabs in the dark. Nevertheless, by looking at the long-term effects of siding with different experts at different times, it was possible to learn to benefit from their domain knowledge. Players who did this outperformed players trying to learn the problem from scratch, as well as players who learned to interact with the experts on a more local scale.

References

Dienes, Z.; Altmann, G.; and Gao, S.-J. 1999. Mapping across domains without feedback: A neural network model of transfer of implicit knowledge. *Cognitive Science* 23:53–82.

Meeden, L. A. 1996. An incremental approach to developing intelligent neural network controllers for robots. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 26:474–485.

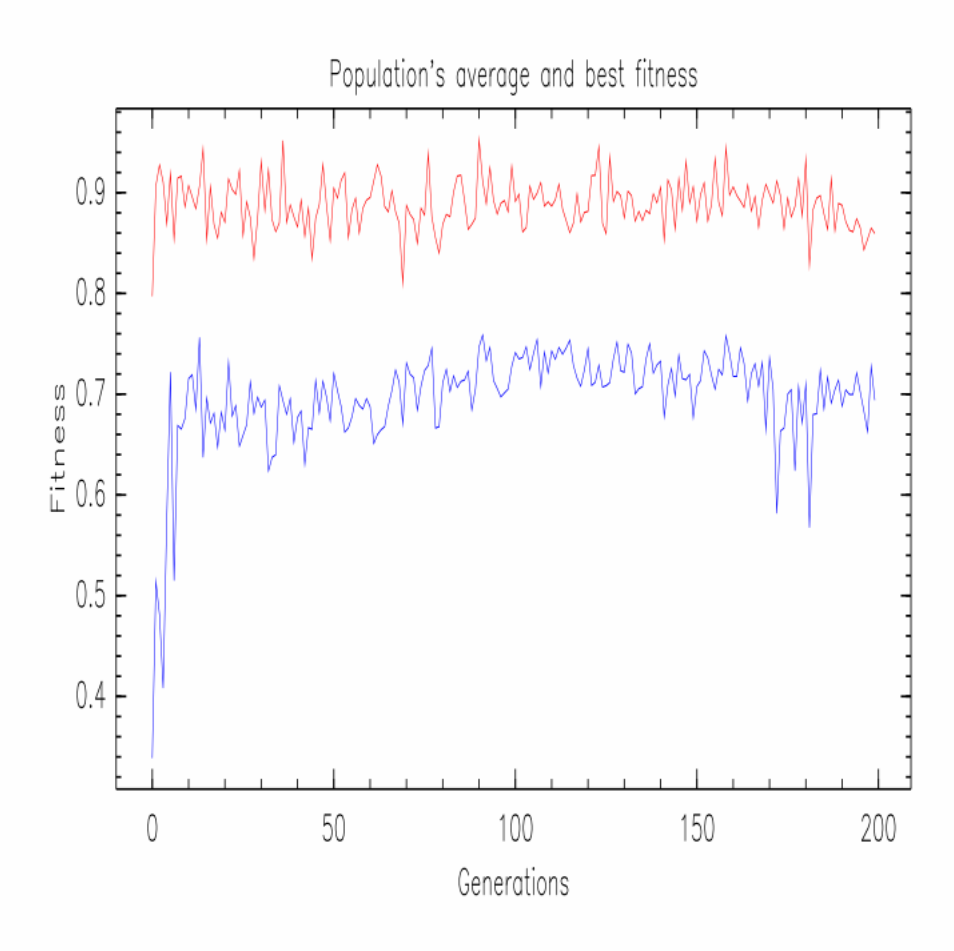


Figure 3: Plot of fitness over the course of a typical NEAT evolution

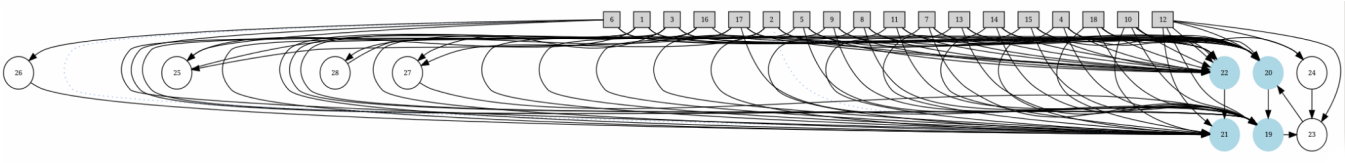


Figure 4: Topology of most successful NEAT chromosome (epoch 187);
note the recurrent relationship between nodes 19, 20, and 23